Inspector-Executor Load Balancing Algorithms for Block-Sparse Tensor Contractions

David Ozog,* Jeff R. Hammond,† James Dinan,† Pavan Balaji,† Sameer Shende,* Allen Malony*

*University of Oregon {*ozog,sameer,malony*}*@cs.uoregon.edu*
†Argonne National Laboratory. {*jhammond, dinan, balaji*}*@anl.gov*

***Abstract*—Good load-balancing methods are required in order to obtain scalability from the NWChem coupled-cluster module, which allows the detailed study of chemical problems by iteratively solving the Schrödinger equation with an accurate ansatz. In this application, a relatively large amount of task information can be obtained at minimal cost, which suggests a static mapping of task groups to processors can be a simple and more efficient alternative to centralized dynamic load balancing. The distributed tensor contractions are block sparse, and an a priori inspection can quickly distinguish non-null tasks and assign them cost estimations based on characteristics such as their dimensions. Architecture-specific and empirically driven performance models of the dominant `SORT` and `DGEMM` routines serve as a cost estimator for a once-per-simulation static partitioning process. In this paper we demonstrate this inspector/executor technique, which improves the NWChem coupled-cluster module's execution time by as much as ∼50% at scale. The technique is applicable to any scientific application requiring load balance where performance models or estimations of kernel execution times are available.**

***Keywords*-Dynamic Load Balancing, Static Partitioning, Quantum Chemistry, Coupled Cluster, Global Arrays, ARMCI, NWChem**

I. Introduction

Load balancing of irregular computations is a serious challenge for petascale and beyond because the growing number of processing elements (PEs)—which now exceeds 1 million on systems such as Blue Gene/Q—makes it increasingly harder to find a work distribution that keeps all the PEs busy for the same period of time. Additionally, any form of centralized dynamic load balancing, such as master-worker or a shared counter (e.g., Global Arrays' `NXTVAL` [19]), becomes a bottleneck. The competition between the need to extract million-way parallelism from applications and the need to avoid load-balancing strategies that have any component that scales with the number of PEs motivates us to develop new methods for scheduling collections of tasks with widely varying cost; the motivating example in this case is the NWChem computational chemistry package. One of the major uses of NWChem is to perform quantum many-body theory methods such as coupled cluster (CC). Popular among chemists are methods such as CCSD(T) and CCSDT(Q) because of their high accuracy at relatively modest computational cost.[1] In these methods, (T) and (Q) refer to perturbative a posteriori corrections to the energy that are highly scalable (roughly speaking, they resemble MapReduce), while the iterative CCSD and CCSDT steps have much more communication and load imbalance. Thus, this paper focuses on the challenge of load balancing these iterative procedures. However, we believe that our approaches can be applied to noniterative procedures as well.

In this paper, we demonstrate that the inspector-executor model (IE) is effective in reducing load imbalance as well as the overhead from the `NXTVAL` dynamic load balancer when this is used. Additionally, we find that the IE is effective when used in conjunction with static partitioning, which is done both with performance models of each task and with empirical measurements. Since CCSD and CCSDT are iterative procedures, the results from the first iteration can be used to improve the task schedule for many subsequent iterations. Our tests with two examples of IE in the NWChem coupled-cluster codes show better application scalability and open the door to new families of load-balancing algorithms as one develops different types of cost models and executor strategies.

[1]The absolute costs of these methods is substantial compared to density-functional theory (DFT), for example, but this does not discourage their use when high accuracy is required.

II. Background

In this section, we describe NWChem, coupled-cluster methods, the Global Arrays programming model, and the Tensor Contraction Engine.

A. NWChem

NWChem [9] is the DOE flagship computational chemistry package, which supports most of the widely used methods across a range of accuracy scales (classical molecular dynamics, ab initio molecular dynamics, molecule density-functional theory (DFT), perturbation theory, coupled-cluster theory, etc.) and many of the most popular supercomputing architectures (InfiniBand clusters, Cray XT and XE, and IBM Blue Gene). Among the most popular methods in NWChem are the DFT and CC methods, for which NWChem is one of the few codes (if not the only code) that support these features for massively parallel systems. Given the steep computational cost of CC methods, the scalability of NWChem in this context is extremely important for real science. Many chemical problems related to combustion, energy conversion and storage, catalysis, and molecular spectroscopy are untenable without CC methods on supercomputers. Even when such applications are feasible, the time to solution is substantial; and even

small performance improvements have a major impact when multiplied across hundreds or thousands of nodes.

B. Coupled-Cluster Theory

Coupled-cluster theory [44] is a quantum many-body method that solves an approximate Schrödinger equation resulting from the CC ansatz,

$$|\Psi_{CC}\rangle = \exp(T)|\Psi_0\rangle, \tag{1}$$

where $|\Psi_0\rangle$ is the reference wavefunction (usually a Hartree-Fock Slater determinant) and $\exp(T)$ is the cluster operator that generates excitations out of the reference. Please see Refs. [11], [6] for more information.

A well-known hierarchy of CC methods exists that provides increasing accuracy at increased computational cost [5]:

$$\begin{aligned}\cdots < CCSD < CCSD(T) < CCSDT \\ < CCSDT(Q) < CCSDTQ < \cdots\end{aligned}$$

The simplest CC method that is generally useful is CCSD [35], has a computational cost of $O(N^6)$ and storage cost of $O(N^4)$, where N is a measure of the molecular system size. The "gold standard" CCSD(T) method [45], [37], [36], [42] provides much higher accuracy using $O(N^7)$ computation but without requiring (much) additional storage. CCSD(T) is a very good approximation to the full CCSDT [33], [47] method, which requires $O(N^8)$ computation and $O(N^6)$ storage. The addition of quadruples provides chemical accuracy, albeit at great computational cost. CCSDTQ [27], [28], [34] requires $O(N^{10})$ computation and $O(N^8)$ storage, while the perturbative approximation to quadruples, CCSDT(Q) [26], [29], [8], [23], reduces the computation to $O(N^9)$ and the storage to $O(N^6)$. Such methods have recently been called the "platinum standard" because of their unique role as a benchmarking method that is significantly more accurate than CCSD(T)[41].

An essential aspect of an efficient implementation of any variant of CC is the exploiting of symmetries, which has the potential to reduce the computational cost and storage required by orders of magnitude. Two types of symmetry exist in molecular CC: spin symmetry [16] and point-group symmetry [10]. Spin symmetry arises from quantum mechanics. When the spin state of a molecule is a singlet, some of the amplitudes are identical; and thus we need store and compute only the unique set of them. The impact is roughly that N is reduced to $N/2$ in the cost model, which implies a reduction of one to two orders of magnitude in CCSD, CCSDT, and CCSDTQ. Point-group symmetry arise from the spatial orientation of the atoms. For example, a molecule such as benzene has the symmetry of a hexagon, which includes multiple reflection and rotation symmetries. These issues are discussed in detail in Refs. [43], [17]. The implementation of degenerate group symmetry in CC is difficult; and NWChem, like most codes, does not support it. Hence, CC calculations cannot exploit more than the 8-fold symmetry of of the D_{2h} group, but this is still a substantial reduction in computational cost.

While the exploitation of symmetries can substantially reduce the computational cost and storage requirements of CC, these methods also introduce complexity in the implementation. Instead of performing dense tensor contractions on rectangular multidimensional arrays, point-group symmetries lead to block diagonal structure, while spin symmetries lead to symmetric blocks where only the upper or lower triangle is unique. This is one reason that one cannot, in general, directly map CC to dense linear algebra libraries. Instead, block-sparse tensor contractions are mapped to BLAS at the PE level, leading to load imbalance and irregular communication between PEs. Ameliorating the irregularity arising from symmetries in tensor contractions is one of the major goals of this paper.

C. Global Arrays

Global Arrays (GA) [31], [32] is a PGAS-like global-view programming model that provides the user with a clean abstraction for distributed multidimensional arrays with one-sided access (put, get and accumulate). GA provides numerous additional functionalities for matrices and vectors, but these are not used for tensor contractions because of the nonrectangular nature of these objects in the context of CC. The centralized dynamic load balancer `NXTVAL` was inherited from TCGMSG, a pre-MPI communication library. Initially, the global shared counter was implemented by a polling process spawned by the last PE, but now it uses ARMCI remote fetch-and-add, which goes through the ARMCI communication helper thread. Together, the communication primitives of GA and `NXTVAL` can be used in a template for-loop code that is general and can handle load imbalance, at least until such operations overwhelm the computation, because of work starvation or communication bottlenecks that emerge at scale. A simple variant of the GA "get-compute-update" template is show in Alg. 1. For computations that are already load balanced, one can use the GA primitives and skip the calls `NXTVAL`, a key feature when locality optimizations are important, since `NXTVAL` has no ability to schedule tasks with affinity to their input or output data. This is one of the major downsides of many types of dynamic load-balancing methods—they lack the ability to exploit locality in the same way that static schemes do.

While there exist several strategies for dynamically assigning collections of tasks to processor cores, most present a trade-off between the quality of the load balance and scaling to a large number of processors. Centralized load balancers can be effective at producing evenly distributed tasks, but they can have substantial overhead. Decentralized alternatives such as work stealing [13], [3] may not achieve the same degree of load balance, but their distributed nature can reduce the overhead substantially.

D. Tensor Contraction Engine

The Tensor Contraction Engine (TCE) [21], [4] is a project to automate the derivation and parallelization of quantum many-body methods such as CC. As a result of this project, the first parallel implementations of numerous methods were

Algorithm 1 The canonical Global Arrays programming template. One can easily generalize this to multidimensional arrays, multiple loops and blocks of data, rather than single elements. As long as the time spent in Foo is greater than that spent in NXTVAL, Get and Update, this is a scalable algorithm.

```
Global Arrays: A, B
Local Buffers: a, b
for i = 1 : N do
  if NXTVAL(my_pe)=True then
    Get A(i) into a
    b = Foo(a)
    Update B(i) with b
  end if
end for
```

created and applied to larger scientific problems than previously possible. The original implementation in NWChem by Hirata was general and required significant tuning to scale CC to more than 1,000 processes in NWChem [24]. The tuning applied to the TCE addressed essentially all aspects of the code, including more compact data representations (spin-free integrals are antisymmetrized on the fly in the standard case), reduction in communication by applying additional fusion that the TCE compiler was not capable of applying, and many other optimizations. In some cases, dynamic load balancing (DLB) was eliminated altogether when the cost of a diagram (a single term in the CC eons) was insignificant at scale and DLB was unnecessary overhead.

The TCE code generator uses a stencil that is a straightforward generalization of Alg. 2. For a term such as

$$Z(i,j,k,a,b,c) += \sum_{d,e} X(i,j,d,e) * Y(d,e,k,a,b,c), \quad (2)$$

which is a bottleneck in the solution of the CCSDT equations, the data is tiled over all the dimensions for each array and distributed across the machine in a one-dimensional global array. Multidimensional global arrays are not useful because they do not support block sparsity or index permutation symmetries. Remote access is implemented by using a lookup table for each tile and a GA Get operation. The global data layout is not always appropriate for the local computation, however; therefore, immediately after the Get operation completes, the data is rearranged into the appropriate layout for the computation. Algorithm 2 gives an overview of a distributed tensor contraction in TCE. The Fetch operation combines the remote Get and local rearrangement for compactness of notation. The SYMM function is a condensation of a number of logical tests in the code that determine whether a particular tile will be nonzero. These tests consider the indices of the tile and not any indices within the tile because each tile is grouped such that the symmetry properties of all its constitutive elements are identical. In Alg. 2, the indices given for the local buffer contraction are the tile indices, but these are merely to provide the ordering explicitly. Each tile index represents a set of contiguous indices so the contraction is between multidimensional arrays, not single elements. However, one can also think of the local operation as the dot product of two tiles.

Algorithm 2 Pseudocode for the default TCE implementation of Eq. 2.

```
Tiled Global Arrays: X, Y, Z
Local Buffers: x, y, z
for all i, j, k ∈ Otiles do
  for all a, b, c ∈ Vtiles do
    if NXTVAL(my_pe)=True then
      if SYMM(i,j,k,a,b,c)=True then
        Allocate z for Z(i,j,k,a,b,c) tile
        for all d, e ∈ Vtiles do
          if SYMM(i,j,d,e)=True then
            if SYMM(d,e,k,a,b,c)=True then
              Fetch X(i,j,d,e) into x
              Fetch Y(d,e,k,a,b,c) into y
              Contract z(i,j,k,a,b,c)+=x(i,j,d,e)*y(d,e,k,a,b,c)
            end if
          end if
        end for
        Accumulate z into Z(i,j,k,a,b,c)
      end if
    end if
  end for
end for
```

III. Inspector/Executor Motivation and Design

In this section we discuss our motivation in implementing an inspector/executor and the design of our cost paritioning strategy for CC simulations in NWChem. While we describe our enhancements to the inspector/executor model by considering performance models for the execution time of `DGEMM` and `SORT4`, this technique can be applied to any collection of performance models for kernels in other scientific applications. We finally discuss the use of inspector/executor static partitioning for load balancing.

A. Inspector/Executor

To evaluate the sparsity of the tensor contractions and schedule the workload efficiently, we collect relevant nonzero task information by breaking the tensor contraction problem into two major components: inspection and execution (similar to Refs. [1], [15], [14]). In its simplest form, the inspector agent loops through relevant components of the parallelized section and collates tasks (Alg. 3). This phase is limited to computationally inexpensive arithmetic operations and conditionals that classify and characterize tasks. Specifically, the first conditional of any particular tensor contraction routine in NWChem evaluates spin and spatial symmetry to determine whether a tile of the tensor contraction has a nonvanishing element [21]. Further along, in a nested loop over common indices, another conditional tests for nonzero tiles of a contraction operand by spin and spatial symmetry. The inspector's

primary advantage is that it distinguishes nonzero tasks and gathers them before entering the dynamic load balancing phase of the application, thereby reducing extraneous calls to NXTVAL.

With a simple inspector in place, we can gather information regarding the sparsity of the entire computation. Consider the graph in Fig. 1, where the yellow bars correspond to the total number of tasks and the red bars correspond to the tasks that result in at least one call to DGEMM as detected by the inspector. The inspector suggests that in CCSD approximately 73% of calls to NXTVAL are unnecessary, and in CCSDT upwards of 95% of calls are also unnecessary.

The motivation for implementing an inspector is that the average time per call to NXTVAL increases with the number of processes. This increase is caused primarily by contention on the memory location of the counter, which performs atomic read-modify-write (RMW) operations (in this case the addition of 1) using a mutex lock. For a given number of total incrementations (i.e., a given number of tasks), when more processes do simultaneous RMWs, on average they must wait longer to access to the mutex. This effect is clearly displayed in a flood-test microbenchmark (Fig. 2) where a collection of processes call NXTVAL several times (without doing any other computation). In this test, only off-node processes are allowed to increment the counter (via a call to ARMCI_Rmw); otherwise, the on-node processes would be able to exploit the far more efficient shared-memory incrementation, which occurs on the order of several nanoseconds.The average execution time per call to NXTVAL always increases as more processes are added.

The increasing overhead of scaling with NXTVAL is also directly seen in performance profiles of the tensor contraction routines in NWChem. For instance, Fig. 3 shows a profile of the mean inclusive time for the dominant methods in a CC simulation of a water cluster with 10 molecules. Notice that the time spent within NXTVAL accounts for about 37% of the entire simulation.

While the following section describes a more complicated version of the inspector, the executor is the same in both cases. The pseudocode for the executor is shown in Alg. 5, where tasks gathered in the inspection phase are simply looped over. Notice that the executor contains the inner loop of the default TCE implementation with another set of symmetry conditionals. These conditionals might at first glance appear to be an opportunity for more NXTVAL eliminations; however, we have found experimentally that non-null tasks rarely enter this inner loop making null calls to the global counter. Therefore, we choose these coarser-grained tasks, a choice that also has the performance benefit of making far fewer calls to the Accumulate function.

B. Task Cost Characterization

Since the average time per call to NXTVAL increases with the number of participating processors, strong scaling improvements will result with the above inspector/executor implementation, which eliminates a large number of extraneous

Algorithm 3 Pseudocode for the inspector used to implement Eq. 2 (simple version).

```
for all i,j,k ∈ Otiles do
  for all a,b,c ∈ Vtiles do
    if SYMM(i,j,k,a,b,c)=True then
      Add Task(i,j,k,a,b,c) to TaskList
    end if
  end for
end for
```

Algorithm 4 Pseudocode for the inspector used to implement Eq. 2 (with cost estimation and static partitioning).

```
for all i,j,k ∈ Otiles do
  for all a,b,c ∈ Vtiles do
    if SYMM(i,j,k,a,b,c)=True then
      Add Task(i,j,k,a,b,c,d,e,w) to TaskList
      cost_w = SORT4_performance_model_estm(sizes)
      for all d,e ∈ Vtiles do
        if SYMM(i,j,d,e)=True then
          if SYMM(d,e,k,a,b,c)=True then
            cost_w = cost_w + ...
            SORT4_performance_model_estm(sizes)
            cost_w = cost_w + ...
            DGEMM_performance_model_estm(m,n,k)
            Compute various SORT4 costs
          end if
        end if
      end for
    end if
  end for
end for
myTaskList = Static_Partition(TaskList)
```

Algorithm 5 Pseudocode for the executor used to implement Eq. 2.

```
for all Task ∈ Tasklist do
  Extract (i,j,k,a,b,c) from Task
  Allocate local buffer for Z(i,j,k,a,b,c) tile
  if SYMM(i,j,d,e)=True then
    if SYMM(d,e,k,a,b,c)=True then
      Fetch X(i,j,d,e) tile into local buffer
      Fetch Y(d,e,k,a,b,c) tile into local buffer
      Z(i,j,k,a,b,c)+=X(i,j,d,e)*Y(d,e,k,a,b,c)
      Accumulate Z(i,j,k,a,b,c) buffer into global Z
    end if
  end if
end for
```

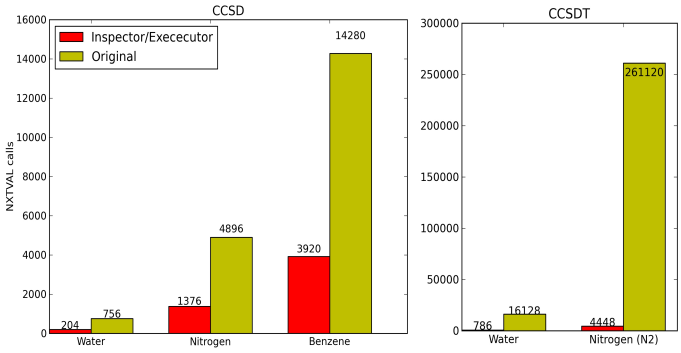

Fig. 1. The total number of calls to NXTVAL in the most time-consuming tensor contraction in CCSD (left) and CCSDT (right). The monomer simulation size increases from left to right. We see that larger simulations generally make more extraneous calls to NXTVAL.

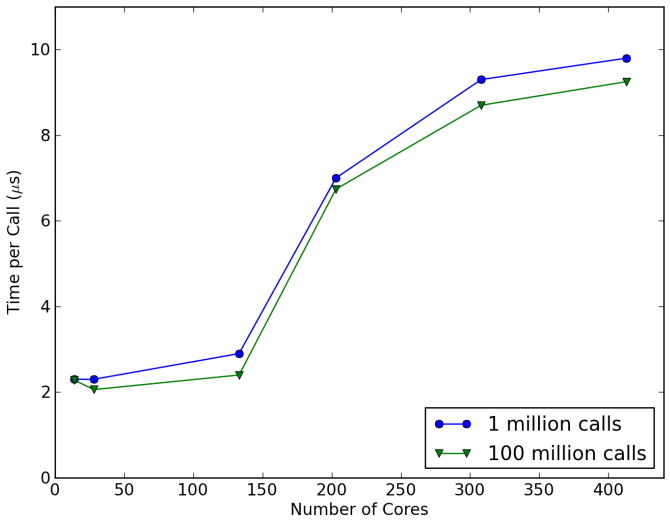

Fig. 2. NXTVAL flood benchmark showing the execution time per call to NXTVAL for 1 million simultaneous calls. Data for a benchmark run with 100 million calls is also shown to emphasize that the curve shape is a feature independent of the number of calls in the benchmark.

calls to the global counter (see Section IV). However, the inspector/executor presented so far still depends on a centralized dynamic load-balancing scheme, which is not ideal for massive scalability. We therefore further develop the inspector/executor model with the intent to eliminate NXTVAL calls from the entire CC module (pseudocode shown in Alg. 4).

By counting the number of FLOPS for a particular tensor contraction (Fig. 4), we see that a great deal of load imbalance is inherent in the overall computation. The centralized dynamic global counter does an acceptable job of handling this imbalance by atomically providing exclusive task IDs to processes that request work. To effectively eradicate the centralization, we first need to estimate the cost of all tasks, then schedule the tasks so that each processor is equally loaded.

In the tensor contraction routines, parallel tile-level matrix multiplications and matrix sorts execute locally within the local memory space of each processor. The key computational

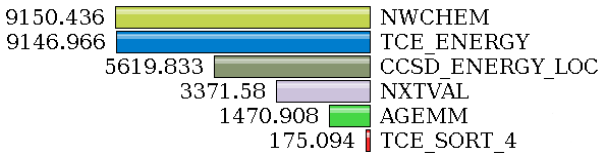

Fig. 3. Average inclusive-time (in seconds) profile of a 14-water monomer CCSD simulation with the aug-cc-PVDZ basis for 861 MPI processes across 123 Fusion nodes connected by InfiniBand. Notice the NXTVAL routine consumes 37% of the entire computation. This profile was made using TAU [38], and for clarity several methods are ignored.

kernels are the DGEMM and the various SORT4 subroutines. The key communication routines are the Global Arrays get (ga_get) and accumulate (ga_acc) methods. On a high-speed switched fabric network such as the Fusion cluster at Argonne National Laboratory, where we conduct most of our experiments (see Section IV), the one-sided RDMA communication operations in NWChem are efficient (e.g., relative to the ARMCI over sockets implementation), and their execution time has negligible variation between tasks. Therefore, our design for static partitioning depends on constructing architecture-specific performance models for the dominant DGEMM and SORT4 routines, where there is large variation in size and thus execution time. These performance models serve as computational cost predictors for the tensor contraction tasks.

1) DGEMM: The DGEMM is a double-precision general matrix multiply within the Basic Linear Algebra Subprograms (BLAS) interface. The model calculates a scaled product of two matrices, A and B, and stores the result in matrix C:

$$C \leftarrow \alpha AB + \beta C,$$

where α and β are the scalar coefficients. An interface to this routine is available in several linear algebra libraries with highly tuned implementations (MKL [22], GotoBLAS2 [18], ATLAS [48]). While the exact run-time characteristics of the DGEMM vary with implementation and architecture, in general the estimated time of operation as a function of the matrix dimensions, $t(m, n, k)$, can be approximated with the following simple performance model:

$$t(m, n, k) = a(mnk) + b(mn) + c(mk) + d(nk), \quad (3)$$

where m is the number of rows in matrix A; n is the number of columns in B, k is the number of columns in A (and rows in B); and a, b, c, and d are coefficients dependent on the system. Respectively, the terms in this model are based on the fact that general matrix multiplication consists of $m * n$ dot products of length k, the corresponding $m*n$ store operations in C, m loads of size k from A, and n loads of size k from B. The coefficients a, b, c, and d are determined by solving a nonlinear least squares problem (with standard methods [30]) in order to characterize the performance of DGEMM on a particular system given the dimensions of the input matrices. Specific details and an example are discussed in Section IV-B.

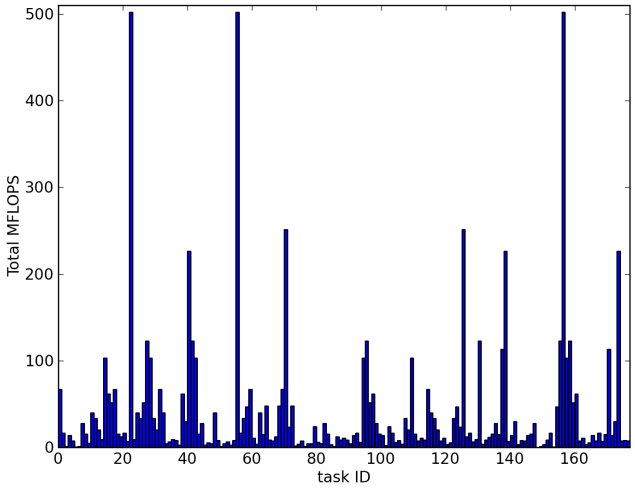

Fig. 4. Total MFLOPS for each task in a single CCSD T_2 tensor contraction for a water monomer simulation. This is a good overall indicator of load imbalance for this particular tensor contraction.

2) SORT4: The other primary kernel in the NWChem CC application is the `SORT4` routine. This function arises from the fact that two binary tensor contractions and an addition are converted into a tensor addition and a contraction by applying the distributive and commutative properties of tensors. Prior to factorization, however, the TCE must sort binary tensor contraction expressions so that two contracted tensors are in a unique order and the external indices are in ascending order across the two tensors (see [20] for details). Each TCE routine will apply some number of transformation that vary in form from one routine to another.

The average performance for the `SORT4` routines is fairly consistent as a function of input size (Fig. 7). However, various types of sort routines depend on the permutations of orbital indices. Figure 7 shows that the sort implementations corresponding to different index permutations show different performance characteristics. Therefore, this form of the `SORT4` (which is most apparent in CCSD computations) requires four performance models, one for each sort type. For this kernel, a simple polynomial cubic fit suffices to capture a reasonable estimation of the sort performance cost for a given input on most modern cache hierarchies; however, such a fit might not work on future architectures. In general, ascertaining a kernel's performance model for a particular system is a matter of discovering a suitable fit function.

C. Static Partitioning

In our static load-balancing algorithm, the inspector applies the `DGEMM` and `SORT4` performance models to each non-null tile encountered, thereby assigning a cost estimation to each task of the tensor contraction. With the collection of weighted tasks, a static partitioning problem must then be solved, scheduling groups of tasks (partitions) to processors in such a way that computational load imbalance is minimized. In general, solving this problem optimally is NP-hard [7], so there is a trade-off between computing an ideal assignment of task partitions and the overhead required to do so. Therefore, our design defers such decisions to a partitioning library (in our case, Zoltan [12]) which gives us the freedom to experiment with load-balancing parameters (such as the balance tolerance threshold) and their effects on the performance of the CC tensor contraction routines.

Currently we employ static block partitioning, which intelligently assigns "blocks" (or consecutive lists) of tasks to processors based on their associated weights (no geometry or connectivity information is incorporated, as in graph/hypergraph partitioning). However, incorporating task connectivity in terms of data locality has been shown to be a viable means of minimizing data access costs [25]. Our technique focuses on balancing the computational costs of task groups as opposed to exploiting their connectivity, which also matters a great deal at scale. Fortunately, our approach is easily extendible to include such data-locality optimizations by solving the partition problem in terms of making ideal cuts in a hypergraph representation of the task-data system. In such a graph, nodes correspond to tasks and hyperedges representing common data elements connect the nodes. The take-away message is that an application making use of the inspector/executor static partitioning technique could presumably partition tasks based on any partitioning algorithm.

IV. Experimental Results

This section provides experimental performance results of several experiments on Fusion, a large InfiniBand cluster at Argonne National Laboratory. Each node has 36 GB of RAM and two quad-core Intel Xeon Nehalem processors running at 2.53 GHz. Both the processor and network architecture are appropriate for this study because NWChem has been heavily optimized for multicore x86 processors and InfiniBand networks. The system is running Linux kernel 2.6.18 (x86_64). NWChem was compiled with GCC 4.4.6, which was previously found to be just as fast as Intel 11.1 because of the heavy reliance on BLAS for floating-point-intensive kernels, for which we employ GotoBLAS2 1.13. The high-performance interconnect is InfiniBand QDR with a theoretical throughput of 4 GB/s per link and 2 μs latency. The communication libraries used were ARMCI from Global Arrays 5.1, which is heavily optimized for InfiniBand, and MVAPICH2 1.7 (NWChem uses MPI sparingly in the TCE).

First, we present an analysis of the strong scaling effects of using `NXTVAL`. Then we describe experiments comparing the original NWChem code with two versions of inspector/executor: one, called I/E Nxtval, that simply eliminates the extraneous calls to `NXTVAL` and one that eliminates all calls to `NXTVAL` in certain methods by using the performance model to estimate costs and Zoltan to assign tasks statically. Because the second technique incorporates both dynamic load balancing and static partitioning, we call it I/E Hybrid.

A. Scalability of centralized load-balancing

The scalability of centralized dynamic load balancing with `NXTVAL` in the context of CC tensor contractions in NWChem

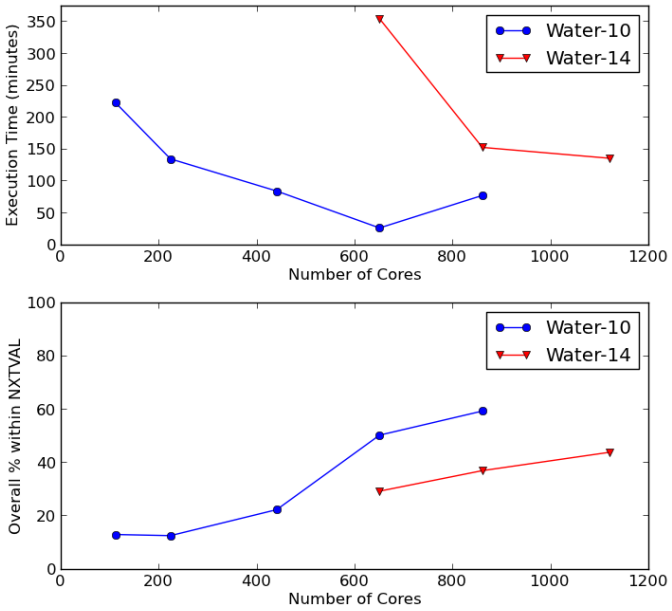

Fig. 5. Total percentage of execution time spent in NXTVAL for a 10-H_2O CCSD simulation (15 iterations) with the aug-cc-pVDZ basis running on the Fusion cluster (without inspector/executor). Notice that w14 will not fit on less than 64 nodes. These data points were extracted from mean inclusive-time profiles as in Fig. 3.

was evaluated by measuring the percentage of time spent incrementing the counter (averaged over all processes) in two water cluster simulations. The first simulation (blue curve in Fig. 5) is a simulation of 10-water molecules using the aug-cc-pVDZ basis, and the second simulation (red curve) is the same but with 14-water molecules. The percentages are extracted from TAU profiles of the entire simulation run, with the inclusive time spent in NXTVAL divided by the inclusive time spent in the application.

Figure 5 shows that the percentage of time spent in NXTVAL always increases as more processors are added to the simulation. This increase is partly because of a decrease in computation per processor, but also because of contention for the shared counter, as displayed in Fig. 2. For 10-water molecules, NXTVAL eventually consumes about 60% of the overall application time as we approach 1,000 processes. In the larger 14-water simulation, NXTVAL consumes only about 30% of the time with 1,000 processes, because of the increase in computation per process relative to the 10-water simulation. The 14-water simulation failed on 63 nodes (441 cores in Fig. 5) because of insufficient memory.

B. Performance Models

While dynamic load balancing is done on the fly and therefore uses the instantaneous conditions of the simulation to decide what to do next, static load balancing (partitioning) requires some estimate of the time spent in each task to determine a reasonable work distribution. To use static partitioning effectively, we developed performance models for the kernels used for every tensor contraction. The first model, DGEMM, is the well-known BLAS3 operation for multiplying two matrices. The second, SORT4, is used by TCE to rearrange tiles of tensors in local memory in order to align the dimensions properly such that they can be contracted using matrix-matrix multiplication. While SORT4 is a bandwidth-intensive operation, its cost is small compared with the loss in efficiency that would arise from the abandonment of DGEMM in favor of unoptimized loop-based tensor contractions. Because the relevant input parameters for DGEMM and SORT4 fit within a known range that is directly related to parameters in an NWChem input file (namely, the maximum size of each tile dimension), the cost of obtaining performance model parameters empirically is insignificant compared with the NWChem computations, so no attempt is made to develop a model from theory alone. Our models were derived from empirical data collected from a variety of CCSD simulations with different basis sets running on Fusion. The models are obviously most relevant to Fusion but are extendible to similar systems. In the end, the empirical cost model derived offline is not critical because we update the task costs to their measured value during the first iteration. However, there is still value in having an offline model because empirical models cannot be used for non-iterative portions of NWChem, such as perturbative triples or quadruples, which we may eventually want to address using static partitioning.

1) DGEMM: Figure 6 shows illustrative performance data of the DGEMM (for water CCSD), which is fitted to the performance model in Eq. 3. The model space is three-dimensional in m, n, k, where each point corresponds to the empirically measured average time to compute. A projection along a particular dimension (in this case, k) provides an idea of the dependence of DGEMM performance on matrix dimensions. To improve visual quality of the histogram, we take a base-2 logarithm of the m, n, and k values of each DGEMM call. The resulting data is then binned, in this case to the nearest integer. When the data is plotted in a 3D space, it shows clear patterns of the DGEMM execution time as a function of matrix dimensions.

A least-squares fit for the DGEMM model $t(m,n,k)$ provides $a = 2.09 \times 10^{-10}$, $b = 1.49 \times 10^{-9}$, $c = 2.02 \times 10^{-11}$, and $d = 1.24 \times 10^{-9}$, which are consistent with the time to execute a single flop, load, and/or store on this processor. The values of b, c, and d are useful only when one or more of m, n, and k are small, since otherwise floating point is clearly the dominant cost of DGEMM. The inequality of c and d, which would be expected from theory, is due to the fact that TCE always uses the TN (T=transpose, N=no transpose) variant of DGEMM and because the input data, which obviously does not include all possible values of m, n, and k, was skewed in favor of larger A matrices because of the specific nature of the CCSD equations.

Although the input data for this fit is a bit noisy, using the model for static assignment has a way of averaging outliers. Moreover, the least squares fit tends to produce values with a lower percentage of error for higher dimensions. On the Fusion cluster, we see a percentage of error of approximately 20% for small DGEMM calls (around $10 \times 10 \times 10$ doubles) and around

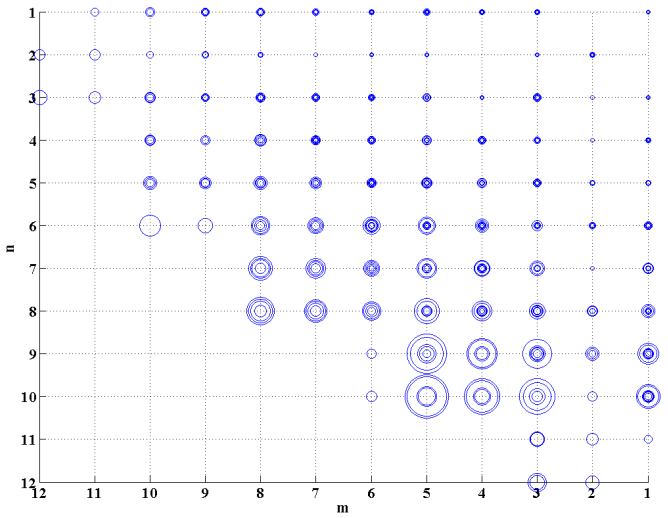

Fig. 6. A $\log_2/\log_2$ projection along the k-axis of a 3D histogram showing the average time spent in calls to DGEMM with dimensions $(2^m, 2^n, 2^k)$ for a water monomer simulation. For example, when m=3 and n=10 in the figure, k has 4 different values corresponding to the four circles drawn at that point. The size of the circles represents the relative average time of DGEMM calls having those approximate dimensions.

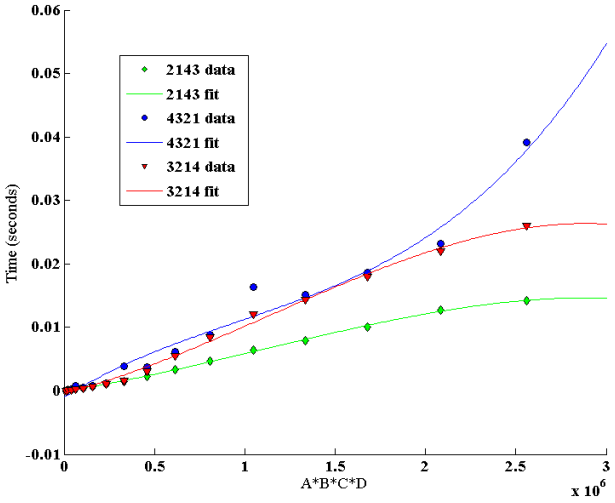

Fig. 7. Example of the GB/s performance of the SORT4 routine fit to a cubic polynomial performance model for a single water molecule. The lower axis is the size of the input in 8-byte words. Each of the three datasets corresponds to various types of sort routines depending on the permutations of orbital indices. Even for NWChem's largest problems this sort will fit in L1/L2 cache, so a cubic polynomial fit suits this kernel.

2% for the largest DGEMM calls (around $10,000 \times 10,000 \times 10,000$)

2) SORT4: We use a cubic polynomial fit of the form

$$p(x) = p_1(x)x^3 + p_2(x)x^2 + p_3x + p_4,$$

where the independent variable x is the product of SORT4 input values A, B, C, and D. As described in Section III, the performance of this kernel depends on the permutation of the input. Figure 7 shows how a different cubic fit for each of the different permutations uniquely characterizes their performance model. For example, after collecting several simulation datasets on Fusion, we arrive at the following fit parameters for the 4321 permutation (the blue curve in Fig. 7): $p_1 = 1.39 \times 10^{-11}$, $p_2 = -4.11 \times 10^{-7}$, $p_3 = 9.58 \times 10^{-3}$, and $p_4 = 2.44$.

Throughout our experiments with SORT4, we have found that the entire operation will fit in L1/L2 cache, and naturally it is a noisy dataset to fit. However, since we are interested mostly in the average computation time of the kernel, it works well for the purposes of load balancing.

C. Inspector/Executor DLB

Applying the I/E Nxtval model to a benzene monomer with the aug-cc-pVTZ basis in the CCSD module results in as much as 33% faster execution of code compared with the original (Fig. 9). The I/E Nxtval version consistently performs about 25-30% faster for benzene CCSD. On the other hand, a nitrogen aug-cc-pVQZ CCSDT simulation, which has high symmetry, performs up to 2.5 times faster than the original code at 280 processor cores (Fig. 8). Above 300 cores, the original code consistently fails on the Fusion InfiniBand cluster with an error in `armci_send_data_to_client()`,

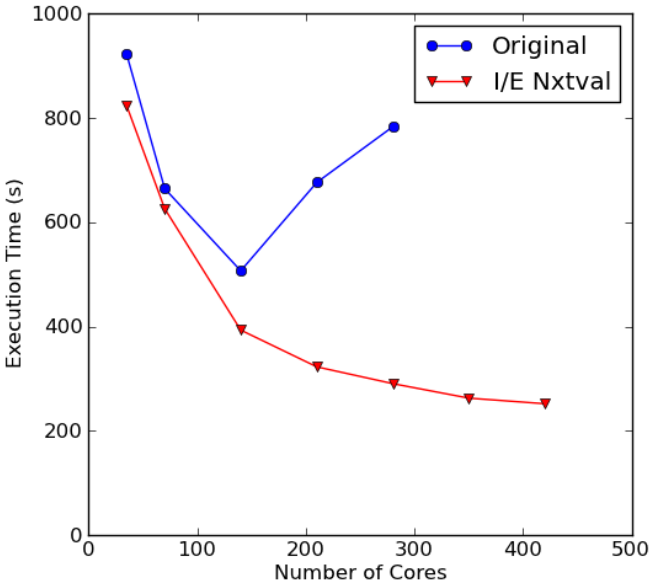

Fig. 8. N_2 aug-cc-pVQZ I/E comparison for a CCSDT simulation.

whereas the I/E Nxtval version continues to scale to beyond 400 processes. This suggests that the error is triggered by an extremely busy NXTVAL server.

D. Static Partition

The I/E Hybrid version applies complete static partitioning using the performance model cost estimation technique to certain tensor contraction methods that are experimentally observed to outperform the I/E Nxtval version. Figure 9 shows that this method always executes in less time than both the original code and the simpler I/E Nxtval version. Though it is not explicitly shown in any of the figures,

TABLE I
300-NODE PERFORMANCE: ORIGINAL CODE FAILS OVER INFINIBAND DUE TO `ARMCI_SEND_DATA_TO_CLIENT()` ERROR

Processes	2400
Nodes	300
I/E Nxtval	498.3 s
I/E Hybrid	483.6 s
Original	-

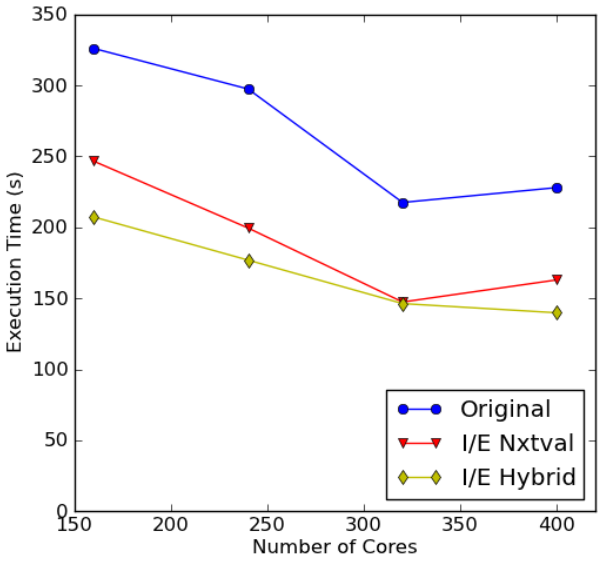

Fig. 9. Benzene aug-cc-pVQZ I/E comparison for a CCSD simulation.

this version of the code also appears to be capable of executing at any number of processes on InfiniBand, whereas the I/E Nxtval and original code eventually trigger the `armci_send_data_to_client()` error.

Unfortunately, it is a difficult feat to transform the machine-generated tensor contraction methods from within the TCE generator, so we have taken a top-down approach where the generated source is changed manually. Because there are over 70 individual tensor contraction routines in the CCSDT module and only 30 in the CCSD module, we currently have I/E Hybrid code implemented only for CCSD.

V. RELATED WORK

Alexeev and coworkers have applied novel static load-balancing techniques to the fragment molecular orbital (FMO) method [2]. FMO differs in computational structure from iterative CC, but the challenge of load balancing is similar, and their techniques parallel the inspector/executor cost estimation model. The FMO system is first split into fragments that are assigned to groups of CPU cores. The size of those groups is chosen based on the solution of an optimization problem, with three major terms representing time that is linearly scalable, nonlinearly scalable, and nonparallel.

Hypergraph partitioning was used by Krishnamoorthy and coworkers to schedule tasks originating from tensor contractions [25]. Their techniques optimize static partitioning based on common data elements between tasks. Such relationships are represented as a hypergraph, where nodes correspond to tasks, and hyperedges (or sets of nodes) correspond to common data blocks the tasks share. The goal is to optimize a partitioning of the graph based on node and edge weights. Their hypergraph cut optimizes load balance based on data element size and total number of operations, but such research lacks a thorough model for representing task weights, which the inspector/executor cost estimation model accomplishes.

The Cyclops Tensor Framework [40], [39] implements CC using arbitrary-order tensor contractions that are implemented by using a different approach from NWChem. Tensor contractions are split into redistribution and contraction phases, where the former permutes the dimensions such that the latter can be done by using a matrix-matrix multiplication algorithm such as SUMMA [46]. Because CTF uses a cyclic data decomposition, load imbalance is eliminated, at least for dense contractions. Point-group symmetry is not yet implemented in CTF and would create some of the same type of load imbalance as seen in this paper, albeit at the level of large distributed contractions rather than tiles. We hypothesize that static partitioning would be highly effective at solving the load-imbalance challenge in CTF.

VI. CONCLUSIONS AND FUTURE WORK

We have presented two improved approaches for conducting load balancing in the NWChem coupled-cluster code generated by TCE. In this application, good load balance was originally achieved by using a global counter to assign tasks dynamically, but application profiling reveals that this method has high overhead, especially when scaling to a large number of processes. Splitting each tensor contraction routine into an inspector and executor component allows us to eliminate unnecessary calls to the counter (from skipped null tasks) and gather relevant cost information regarding tasks, which can be used for static partitioning. We have shown that simply removing extraneous calls to the global counter can dramatically improve the performance of the entire NWChem coupled cluster application, and in some cases it even allows us to scale out to a number of processes that previously was impossible because of the instability of the `NXTVAL` server with large numbers of calls and processes.

The technique of generating performance models for `DGEMM` and `SORT4` to estimate costs associated with load balancing is general to all compute-kernels and can be applied to applications that require large scale parallel task assignment. While other non-centralized dynamic load balancing methods (such as work stealing and resource sharing) could potentially outperform such static partitioning, such methods tend to be difficult to implement and may have centralized components. The approach of using a performance model and a partitioning library together to achieve load balance is easily parallelizable (though in NWChem tensor contractions, we have found a sequential version to be faster because of the inexpensive computations in the inspector) and easy to implement and requires few changes to the original application code.

Because the technique is readily extendible, we plan to improve our optimizations by adding functionality to the inspector. For example, we can exploit proven data locality techniques by representing the relationship of tasks and data elements with a hypergraph and decomposing the graph into optimal cuts [25].

ACKNOWLEDGMENTS

This research used resources of the Argonne Leadership Computing Facility and Laboratory Computing Resource Center at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

The research at the University of Oregon was supported by grants from the U.S. Department of Energy, Office of Science, under contracts DE-FG02-07ER25826, DE-SC0001777, and DE-FG02-09ER25873.

REFERENCES

[1] Gagan Agrawal, Alan Sussman, and Joel Saltz. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. *IEEE Transactions on Parallel and Distributed Systems*, 6:747–754, 1995.

[2] Yuri Alexeev, Ashutosh Mahajan, Sven Leyffer, Graham Fletcher, and Dmitri Fedorov. Heuristic static load-balancing algorithm applied to the fragment molecular orbital method. *Supercomputing*, 2012.

[3] Humayun Arafat, P. Sadayappan, James Dinan, Sriram Krishnamoorthy, and Theresa L. Windus. Load balancing of dynamical nucleation theory monte carlo simulations through resource sharing barriers. In *IPDPS*, pages 285–295, 2012.

[4] Alexander A. Auer, Gerald Baumgartner, David E. Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert Harrison, Sriram Krishnamoorthy, Sandhya Krishnan, Chi-Chung Lam, Qingda Lu, Marcel Nooijen, Russell Pitzer, J. Ramanujam, P. Sadayappan, and Alexander Sibiryakov. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics*, 104(2):211–228, 2006.

[5] R.J. Bartlett. Coupled-cluster approach to molecular structure and spectra: a step toward predictive quantum chemistry. *Journal of Physical Chemistry*, 93(5):1697–1708, 1989.

[6] R.J. Bartlett and M. Musiał. Coupled-cluster theory in quantum chemistry. *Reviews of Modern Physics*, 79(1):291–352, 2007.

[7] S. H. Bokhari. On the mapping problem. *IEEE Trans. Comput.*, 30(3):207–214, March 1981.

[8] Yannick J. Bomble, John F. Stanton, Mihály Kállay, and Jürgen Gauss. Coupled-cluster methods including noniterative corrections for quadruple excitations. *Journal of Chemical Physics*, 123(5):054101, 2005.

[9] E. J. Bylaska, W. A. de Jong, N. Govind, K. Kowalski, T. P. Straatsma, M. Valiev, D. Wang, E. Aprà, T. L. Windus, J. Hammond, J. Autschbach, P. Nichols, S. Hirata, M. T. Hackler, Y. Zhao, P-D Fan, R. J. Harrison, M. Dupuis, D. M. A. Smith, J. Nieplocha, V. Tipparaju, M. Krishnan, A. Vazquez-Mayagoitia, Q. Wu, T. Van Voorhis, A. A. Auer, M. Nooijen, L. D. Crosby, E. Brown, G. Cisneros, G. I. Fann, H. Früchtl, J. Garza, K. Hirao, R. Kendall, J. A. Nichols, K. Tsemekhman, K. Wolinski, J. Anchell, D. Bernholdt, P. Borowski, T. Clark, D. Clerc, H. Dachsel, M. Deegan, K. Dyall, D. Elwood, E. Glendening, M .Gutowski, A Hess, J. Jaffe, B. Johnson, J. Ju, R. Kobayashi, R. Kutteh, Z. Lin, R. Littlefield, X. Long, B. Meng, T. Nakajima, S. Niu, L. Pollack, M. Rosing, G. Sandrone, M. Stave, H. Taylor, G. Thomas, J. van Lenthe, A. Wong, and Z. Zhang. NWChem, a computational chemistry package for parallel computers, version 5.1.1, 2009.

[10] F.A. Cotton. *Chemical Applications of Group Theory*. John Wiley & Sons, 2008.

[11] T. D. Crawford and H. F. Schaefer III. An introduction to coupled cluster theory for computational chemists. In K. B. Lipkowitz and D. B. Boyd, editors, *Reviews in Computational Chemistry*, volume 14, chapter 2, pages 33–136. VCH Publishers, New York, 2000.

[12] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002.

[13] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 53:1–53:11, New York, 2009. ACM.

[14] Stephen Fink and Scott Baden. Runtime support for multi-tier programming of block-structured applications on smp clusters. In Yutaka Ishikawa, Rodney Oldehoeft, John Reynders, and Marydell Tholburn, editors, *Scientific Computing in Object-Oriented Parallel Environments*, volume 1343 of *Lecture Notes in Computer Science*, pages 1–8. Springer Berlin / Heidelberg, 1997.

[15] Stephen J. Fink, Scott B. Baden, and Scott R. Kohn. Efficient run-time support for irregular block-structured applications. *Journal of Parallel and Distributed Computing*, 50(1–2):61–82, 1998.

[16] J. Fuchs and C. Schweigert. *Symmetries, Lie Algebras and Representations: A Graduate Course for Physicists*. Cambridge University Press, 2003.

[17] Jürgen Gauss, John F. Stanton, and Rodney J. Bartlett. Coupled-cluster open-shell analytic gradients: Implementation of the direct product decomposition approach in energy gradient calculations. *Journal of Chemical Physics*, 95(4):2623–2638, 1991.

[18] K. Goto. GotoBLAS, 2007.

[19] Robert J. Harrison. Portable tools and applications for parallel computers. *International Journal of Quantum Chemistry*, 40(6):847–863, 1991.

[20] So Hirata. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *Journal of Physical Chemistry A*, 107:9887–9897, 2003.

[21] So Hirata. Tensor Contraction Engine: Abstraction and Automated Parallel Implementation of Configuration-Interaction, Coupled-Cluster, and Many-Body Perturbation Theories. *Journal of Physical Chemistry A*, 107(46):9887–9897, 2003.

[22] Intel. Intel math kernel library.

[23] Mihály Kállay and Jürgen Gauss. Approximate treatment of higher excitations in coupled-cluster theory. *Journal of Chemical Physics*, 123(21):214105, 2005.

[24] Karol Kowalski, Jeff R. Hammond, Wibe A. de Jong, Peng-Dong Fan, Marat Valiev, Dunyou Wang, and Niranjan Govind. Coupled cluster calculations for large molecular and extended systems. In Jeffrey R. Reimers, editor, *Computational Methods for Large Systems: Electronic Structure Approaches for Biotechnology and Nanotechnology*. Wiley, 2011.

[25] Sriram Krishnamoorthy, Umit Catalyurek, Jarek Nieplocha, and Atanas Rountev. Hypergraph partitioning for automatic memory hierarchy management. In *Supercomputing (SC06)*, 2006.

[26] Stanislaw A. Kucharski and Rodney J. Bartlett. Coupled-cluster methods that include connected quadruple excitations, T_4: CCSDTQ-1 and Q(CCSDT). *Chemical Physics Letters*, 158(6):550–555, 1989.

[27] Stanislaw A. Kucharski and Rodney J. Bartlett. Recursive intermediate factorization and complete computational linearization of the coupled-cluster single, double, triple, and quadruple excitation equations. *Theoretical Chemistry Accounts: Theory, Computation, and Modeling (Theoretica Chimica Acta)*, 80:387–405, 1991.

[28] Stanislaw A. Kucharski and Rodney J. Bartlett. The coupled-cluster single, double, triple, and quadruple excitation method. *Journal of Chemical Physics*, 97(6):4282–4288, 1992.

[29] Stanislaw A. Kucharski and Rodney J. Bartlett. An efficient way to include connected quadruple contributions into the coupled cluster method. *Journal of Chemical Physics*, 108(22):9221–9226, 1998.

[30] D. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):431–441, 1963.

[31] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: a portable "shared-memory" programming model for distributed memory computers. In *Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 340–349, New York, 1994. ACM.

[32] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: A non-uniform-memory-access programming model for

high-performance computers. *The Journal of Supercomputing*, 10:10–197, 1996.

[33] J. Noga and R.J. Bartlett. The full CCSDT model for molecular electronic structure. *Journal of Chemical Physics*, 86(12):7041–7050, 1987.

[34] Nevin Oliphant and Ludwik Adamowicz. Coupled-cluster method truncated at quadruples. *The Journal of Chemical Physics*, 95(9):6645–6651, 1991.

[35] G.D. Purvis III and R.J. Bartlett. A full coupled-cluster singles and doubles model: The inclusion of disconnected triples. *Journal of Chemical Physics*, 76(4):1910–1918, 1982.

[36] Krishnan Raghavachari, John A. Pople, Eric S. Replogle, and Martin Head-Gordon. Fifth-order Møller-Plesset perturbation theory: comparison of existing correlation methods and implementation of new methods correct to fifth order. *Journal of Physical Chemistry*, 94:5579–5586, 1990.

[37] Krishnan Raghavachari, Gary W. Trucks, John A. Pople, and Martin Head-Gordon. A fifth-order perturbation comparison of electron correlation theories. *Chemical Physics Letters*, 157:479–483, May 1989.

[38] Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[39] Edgar Solomonik. Cyclops Tensor Framework. http://www.eecs.berkeley.edu/ solomon/cyclopstf/index.html.

[40] Edgar Solomonik, Jeff Hammond, and James Demmel. A preliminary analysis of Cyclops Tensor Framework. Technical Report UCB/EECS-2012-29, EECS Department, University of California, Berkeley, March 2012.

[41] John Stanton. This remark is attributed to Devin Matthews.

[42] John F. Stanton. Why CCSD(T) works: A different perspective. *Chemical Physics Letters*, 281:130–134, 1997.

[43] John F. Stanton, Jürgen Gauss, John D. Watts, and Rodney J. Bartlett. A direct product decomposition approach for symmetry exploitation in many-body methods, I: Energy calculations. *Journal of Chemical Physics*, 94(6):4334–4345, 1991.

[44] Jiří Čížek. On the correlation problem in atomic and molecular systems. calculation of wavefunction components in Ursell-Type expansion using Quantum-Field theoretical methods. *Journal of Chemical Physics*, 45(11):4256–4266, December 1966.

[45] Miroslav Urban, Jozef Noga, Samuel J. Cole, and R.J. Bartlett. Towards a full CCSDT model for electron correlation. *Journal of Chemical Physics*, 83(8):4041–4046, 1985.

[46] R. A. Van De Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[47] John D. Watts and R.J. Bartlett. The coupled-cluster single, double, and triple excitation model for open-shell single reference functions. *Journal of Chemical Physics*, 93(8):6104–6105, 1990.

[48] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, pages 1–27, Washington, DC, 1998. IEEE Computer Society.